UR10 Performance Analysis

Ravn, Ole; Andersen, Nils Axel; Andersen, Thomas Timm

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

UR10 Performance Analysis

by Asger Winther-Jørgensen
& Christian Ahlburg Dirksen

Technical University of Denmark
Institute for Automation and Control
Ole Ravn
Nils Axel Andersen
Thomas Timm Andersen
29. juni 2014

Contents

1 Introduction

While working with the UR-10 robot arm, it has become apparent that some commands have undesired behaviour when operating the robot arm through a socket connection, sending one command at a time. This report is a collection of the results optained when testing the performance of the different commands available in URScript to control the robot. It will also describe the different time delays discovered when using the UR-10 robot arm.

2 Real-time Interface

Glossary for the following sections:
Controller:
The program running on the UR-10's internal computer, broadcasting robot arm data, recieving and interpretting commands and controlling the arm accordingly.
Interfacing program or program:
A program running on an arbitrary computer, connecting (interfacing) with the controller over a TCP Connection to the controller's 'real time'- or MATLAB-Interface port.

2.1 Congestion Control - Nagle's algorithm?

The first thing that becomes evident when looking at the Matlab interface output of the UR-10 robot is that it appears to have some sort of anti-congestion algorithm in place.

Figures 2.1 and 2.2 show a joint angle as a function of time. In each of the figures, the two series are the same joint-variables plotted as functions of two different times. The red series is the joint angle as a function of the timestamp of the package in which it was broadcast. The blue series is the very same joint angle data series, but this time plotted as a function of the computer localtime at which the package was recieved by the program communicating with the UR Matlab interface.

In figure 2.1 the program was run on a remote computer communicating with the robot arm's internal computer over Ethernet. In figure 2.2 the program was run on the robot arm's internal computer.

Especially in figure 2.2, it is very visible that the packages are not broadcast at a rate of 125Hz, but rather at some other frequency, in this case 5 packages at a time at 25Hz, approximately. For this test, the program interfacing with the Matlab interface was run on the robot arm's internal computer.

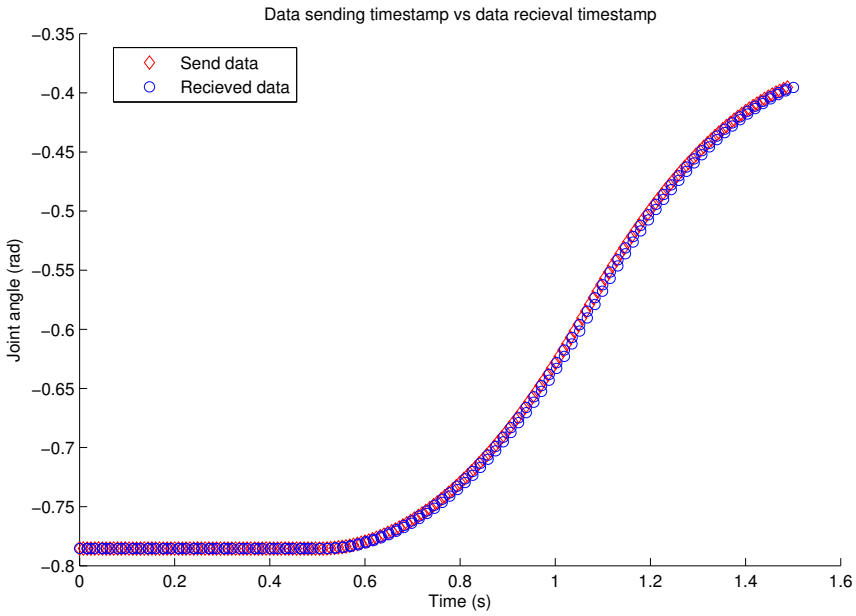

Figure 2.1: Angle values from data packages plotted as a function of package timestamp and as a function of package arrival time when the program was run on a remote computer. It can be seen that the packages arrive at the computer two at a time.

NB: These plots do not in any way tell us anything about the time delay between the controller sending a package and the interfacing program recieving it. The controller timestamp and the program timestamp are on different time scales, but have been translated to the same range for the sole purpose of being displayed in one graph. The translation is arbitrary.

The 25Hz, or 40ms, is the standard acknowledge timeout for UNIX based systems, and Nagle's algorithm will always wait for acknowledgement before sending the next package, so this again points towards the Nagle's algorithm being the problem.

In both Python and C++, a socket parameter can be set, TCP_QUICKACK, which will force any pending acknowledgements to be sent immediately. Setting this flag every time a data package is recieved will make the Nagle's Algorithm on the controller socket send the next package when it becomes available. This solves the problem of packages being bundled for delayed shipment.

This is a work-around of the problem, allowing the Matlab-interface to be used for real-time control in spite of the Nagle-algorithm. It is noteworthy that any closed-loop control of the robot arm automatically implements this

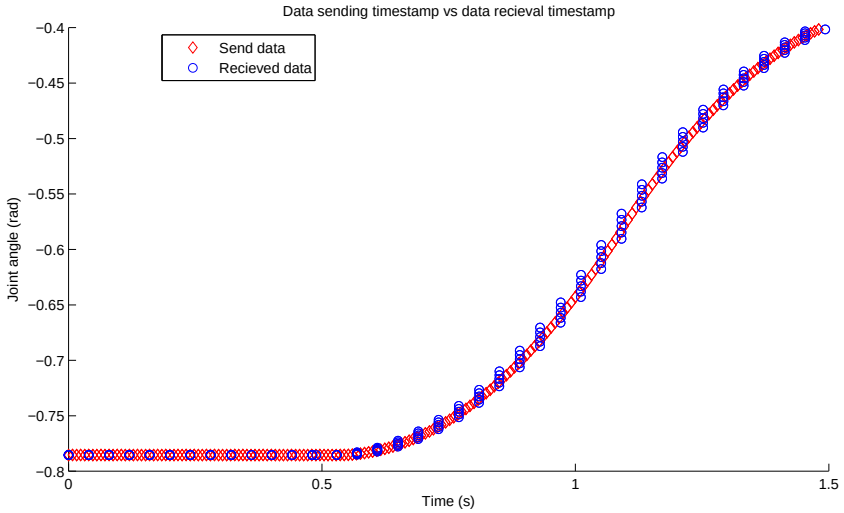

Figure 2.2: Angle values from data packages plotted as a function of package timestamp and as a function of package arrival time when the program was run on the robot arm's internal computer.

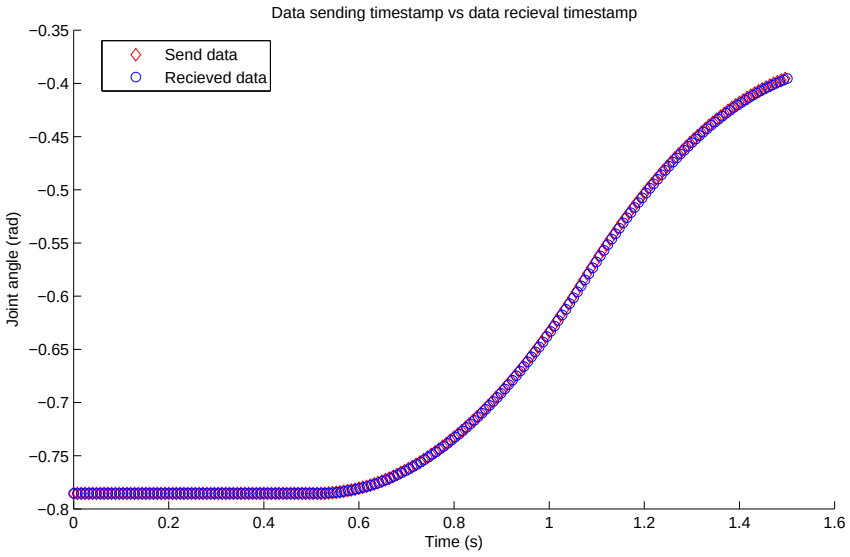

Figure 2.3: Angle values from data packages plotted as a function of package timestamp and as a function of package arrival time when the program included forced immediate acknowledgement. The results are similar for the robot arm's internal computer and the remote computer.

work-around since sending a command as a reaction to the last recieved package will also send the acknowledgement so that the next package will appear on time.

2.2 Sample Frequency

Now that the recieving frequency has been determined to be 125Hz, it would be good to know the frequency with which one can send commands to the robot arm and expect them to be handled on time. To see if this frequency matched the broadcasting frequency, a sampled sinus curve was sent to the robot arm, one sample pr. recieved data package from the robot arm. Figure 2.4 displays the result of a test in which a sinusoidal velocity curve for one joint was sent to the robot arm at the same frequency the robot arm broadcasts with, achieved by sending each sample as a reaction to a recieved package. Here the code used was:

```
starttime = time.now
while time.now < 2*pi+starttime:
  on recieved_new_data:
    qd = [0,0,0,0,0,0]
    qd[4] = sin(10*time.now)
    speedj(qd,5,0.02)
end
```

What can be seen on the figure is that the robot arm's internal target velocity follows, with some delay, the same trajectory as is sent to it, that is it does not undersample or drop packages.
Something else to notice is the sometimes serrated nature of the controller target velocity curve. It seems that commands are lost one sample, but made up for in the next. This suggests that the controller handles input and output asynchronously. This serration also shows that, even though these commands are sent as soon as possible after recieving a package, one cannot be sure that a reaction will be seen until two samples later.

Looking at figure 2.5 we see a different behaviour. In this case, the same sinusoidal curve was sent to the robot arm, but at twice the frequency, that is both the frequenuncy of the sinusoidal curve and the sample frequency were doubled. The code used here was:

```
starttime = time.now
while time.now < 2*pi+starttime:
  qd = [0,0,0,0,0,0]
  qd[4] = sin(11*time.now)
```

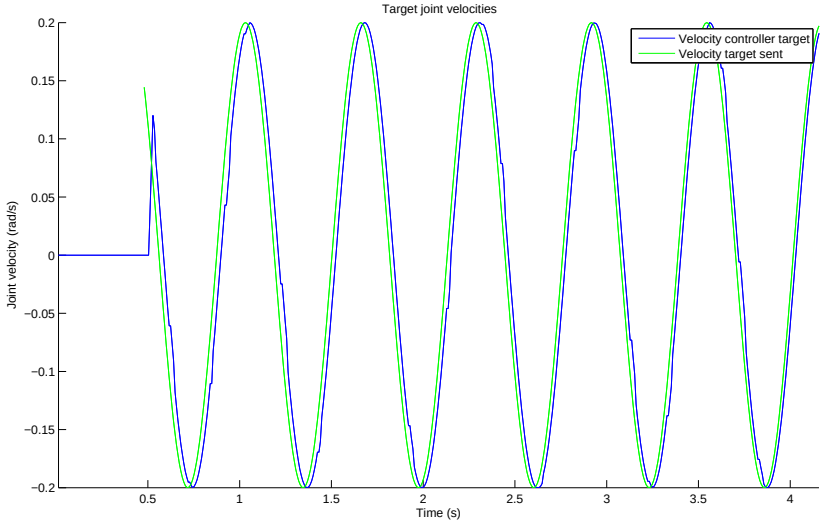

Figure 2.4: Sent target, the sinus shaped velocity curve sent to the robot arm plotted next to the recieved target velocity of the robot arm joint. The sent target is issued at the same frequency as the controller broadcasts

```
  sleep(0.004)
  speedj(qd,5,0.02)
end
```

Misbehaviour was expected in this test, the hope being that samples would be dropped and that the robot target velocity would be an undersampled version of the sent target velocity. This was not the case. Instead, we saw every sample in the robot, but executed one for each controller sample without regard to the time at which it was recieved. It was also observed that the robot arm continued moving after the program was terminated.
From this, it can be deduced that the controller buffers any recieved commands and executes the oldest command, if any, in the buffer at each controller iteration. This can be desirable as no control commands are overlooked, but it does have drawbacks. Doing real-time control on the robot arm, any other command issued to the arm will cause delay in the real-time control if not accounted for by skipping one sample of the control.

When viewing results of long tests, it seems that the timescales used by the robot controller and the computers respectively do not agree on the length of a second. Plotting dataseries as a function of controller timestamp and of computer timestamp on top of each other showed that the computer

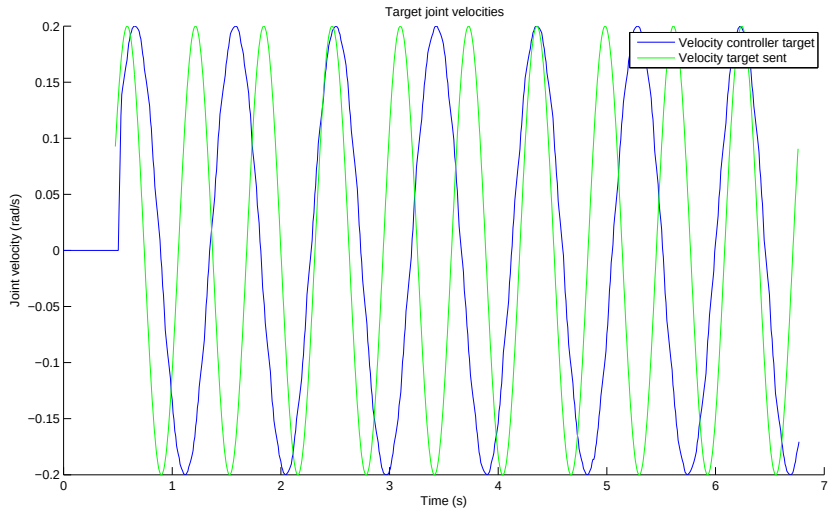

Figure 2.5: Sent target, the sinus shaped velocity curve sent to the robot arm plotted next to the recieved target velocity of the robot arm joint. The sent target is issued at approximately twice the frequency of the controller broadcast

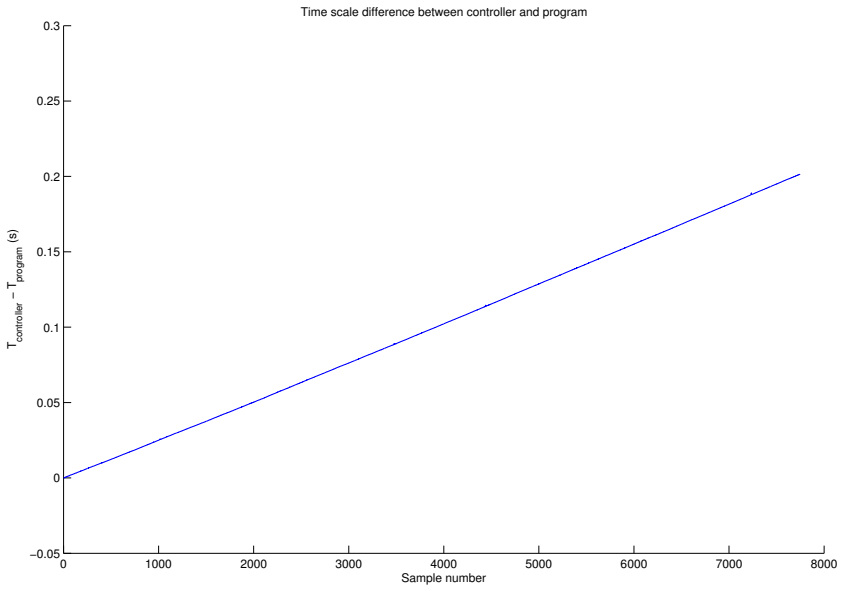

Figure 2.6: Illustration of the timescale difference between the controller time and the program time. Timescales are offset to zero at sample one.

time seemed dialated compared to controller time. This is shown in figure 2.6. Running long tests and comparing last and first package timestamps in controller- and computertime showed the same. The meassured difference between the time intervals was: $\Delta t_{controller} = 0.9969 \cdot \Delta t_{computer}$
This is not a big problem, one just has to account for this when creating controllers and make sure that it is designed for the actual sample time observed.

2.3 Calculation time and delays

Time delays are to be expected when working across networks. It is desirable to determine these delays, as they have a major impact on the actual bandwidth limitations of any controller implementations.
This is adressed in this section.

For a quick estimate of computer-to-computer communication, tests were run in which packages of sizes similar to the commands and data packages were pinged back and forth, and these indicated a maximum response time of 1ms.

Through a series of tests in which a command was sent to the controller from a remote computer and the time was meassured before a reaction could be seen from the controller (The round trip time). The best-case scenario was used when the command was sent immediately after recieving a package (no calculation time). A reaction from the controller, a change in the controller's target velocity, was observed either 8ms later (one sample) or 16ms (two samples).
In all cases, pysical change was meassured by the joint variable encoders in the next sample after controller target velocities had been set, giving a total round-trip time for the signal of 24ms.

To estimate the effect of calculation time before a reaction to a packate is sent, another sinus curve velocity is sent to the robot arm. The velocity command sending is triggered by the arrival of a data package. This time, however, a delay is induced between the package is recieved and the command is sent. In the beginning of the sinus curve, the delay is as short as possible, increasing by 0.000016 per sample until the delay reaches 0.008, a full sample. The code used was:

```
while delay < 0.008:
  on recieved_new_data:
    qd = [0,0,0,0,0,0]
    qd[4] = qd[4] + 0.2*sin(10*time.now)
    sleep(delay)
```

```
    delay += 0.008/500
    speedj(qd,5,0.02)
  sleep(0.00001)
end
```

One resulting graph is shown in 2.7. It can be seen that the target velocity graph is serrated in the beginning of the graph, indicating that the commands do not consistently take the same amount of samples before reaching the controller, as was seen when meassuring reaction times for immediate responses. This serration is caused mainly by network and calculation time, and it is suspected that the controller interacts with network data and the robot asynchronious. Upon reaching a certain delay, the commands consistently reach the target velocity of the controller in two samples. Serration shows again at a 6ms delay, indicating that at a calculation time of less than 6ms will ensure that a command reaches the controller in maximum 2 samples (16ms), and that physical change can be seen within 3 samples (24ms).

This gives a total of somewhere between 16ms and 24ms from a command is issued until a physical change can be observed in the system. This results in a bandwidth limit of approximately 50Hz induced by the delays in communication in the best-case scenario where calculation time is less than 6ms and the robot arm and controlling computer are on the same local wired network.

These results are valid for programs run on a remote computer connected over ethernet and for the robot arm's internal computer alike.

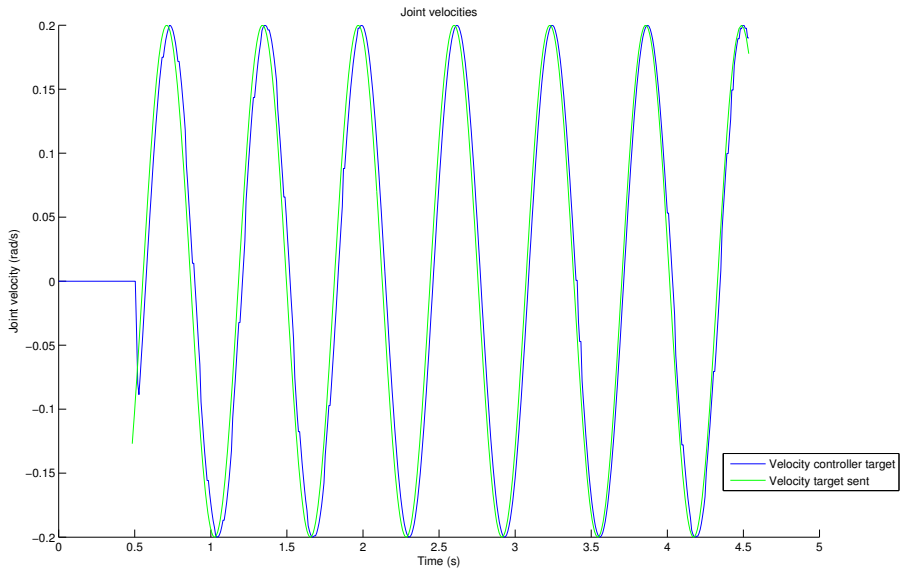

Figure 2.7: Target velocity of the robot controller and the velocity sent in commands as a function of time. For each command sent, the delay until the next command is increased slightly.

3 Commands

The UR script provides a series of commands with which to control the robot arm. Each have their uses and caveats. Some tests of a selection of these commands have been run, and some of these problems have been documented. Table 3.1 shows a list of the tested commands, short descriptions of them and a brief summary of the test results in good/bad form. Extended explanations of the tests and results follow where they have been deemed worthwhile.

Command	Description	Good	Bad
movec	Move along circular arc segment from current pose, through via pose to end pose.	Easy way to make soft motions/curves in cartesian space, for example around obstacles.	Jerks to a halt in final position. Jerks when interrupted.
movej	Simple joint space move command.	No singularity problems.	Jerks when interrupted, this can cause Torque Limit Violation
movel	Simple cartesian space move command.	Movement in cartesian space is easy to plan.	Singularity problems. Jerks when interrupted, can cause Torque limit violation.
servoj	Servo to joint coordinates without pathplanning.	Same as movej	Jerks to a halt in final position. Jerks when interrupted, this can cause Torque Limit Violation.
servoc	Servo to cartesian coordinates without pathplanning	Same as movel	Jerks to a halt in final position. Jerks when interruptet, this can cause Torque Limit Violation.
speedj	Set Joint speeds with desired acceleration.	Works as expected. Can be issued at 125Hz. Will not jerk if interrupted.	Nothing observed.
speedl	Set TCP speed in cartesian space with desired joint acceleration.	Possible to set speeds in cartesian space rather than joint space.	Jerks when interrupted, this can cause Torque Limit Violation. Jerks occur when interrupting movement with a new speed.
stopj	Decellerates joint speeds to zero.	Works as expected.	If given close to the end of a movement, it can result in an overshoot or an undershoot.
stopl	Decellerates TCP speed to zero.	Works as expected.	If given close to the end of a movement, it can result in an overshoot or an undershoot.

Tabel 3.1: List of commands and their caveats

3.1 The movec command

The code for this test:

```
sleep(0.5)
movec(p_via ,p_end ,2,0.6)
sleep(5)
```

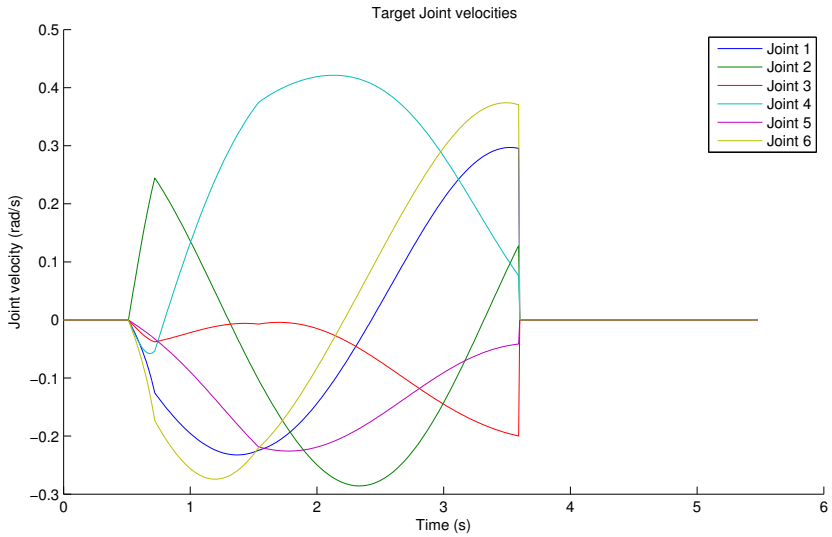

Figure 3.1: Target joint velocities during a movec command execution.

In figures 3.1 and 3.2 the behaviour of movec can be seen.
When the movec reaches it's final pose, the robot arm jerks violently to a halt, inducing notable vibrations in the robot setup. Looking at the target velocity curve, one could expect the target accelerations to be off the charts, similar to when move commands in general are interrupted, but this does not occur according to figure 3.2. It seems that the controller just abandons the robot to it's own devices once the goal position is reached.
The lack of deceleration is an issue, but the lack of effort to break instantly does save the system from joint torque limit violation.

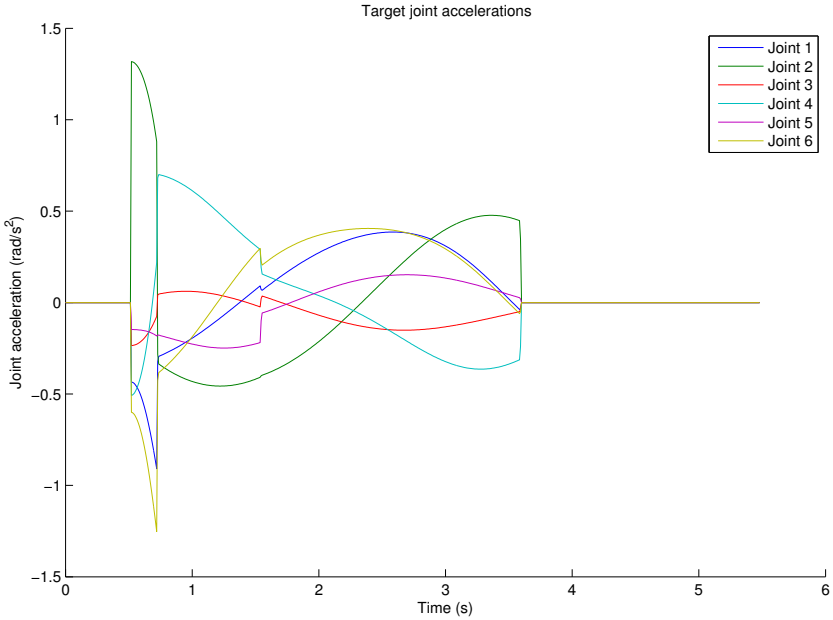

Figure 3.2: Target joint accelerations during a movec command execution.

3.2 The movej command

The code for this test:

```
sleep(0.5)
movej(q1)
sleep(1.2)
movej(q2)
sleep(1.3)
movej(q3)
sleep(5)
```

In figures 3.3 and 3.4, some results of testing the movej command are shown. In this test, three movej commands were sent to the robot arm. The first is sent at 0.5 sec when the robot arm is standing still. The second command is sent at 1.7 seconds, just before the first move is finished. The last command is sent at 3 seconds, midway through the seccond command's execution.
The result is jerky. In itself, the movej command executes a nice, smooth trapezoid trajectory in configuration space, but if it is interrupted by a second movej command before completing, things go wrong.
One likely reason for this would be the controller calculating a new trapezoid from current position upon recieving the interrupting movej command, but

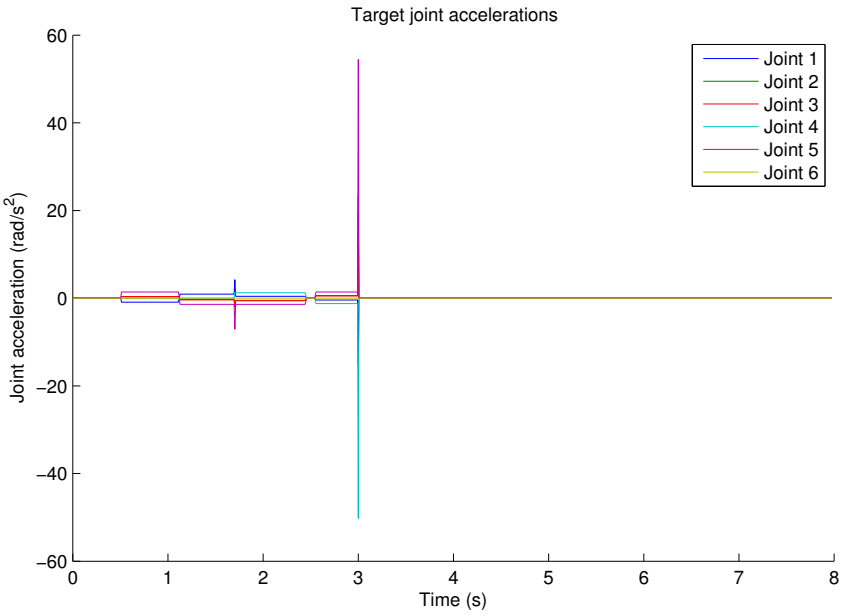

Figure 3.3: Target joint accelerations during a series of overlapping movej commands. One movej command is issued at 0.5 seconds, the next at 1.7s, just before the first finishes, and the third is issued at 3s, way before the second command has finished. Both interruptions lead to acceleration spikes and the second leads to joint torque limit violation.

doing so assuming a still-standing robot. Setting target velocity to something fitting for a still-standing robot when the robot is moving would require great accelerations, as are observed.
As a consequence of this desired great acceleration, large currents are induced over the motors, in extreme cases triggering the joint torque violation safety limit. In this case, a current of over 15A is observed in the shoulder joint motor. These problems could be solved, it seems, by calculating trajectories for move commands not using just current position of the robot, but using the full current robot state including velocities.
Something else of interest observed here is that joint torque limits are not set on the torque caused by interaction with external objects, but total torque due to internal and external torques combined. This is of course the most cost-effective and error robust implementation, but it does force the user to have to consider the joint torque limits when planning sudden stops.

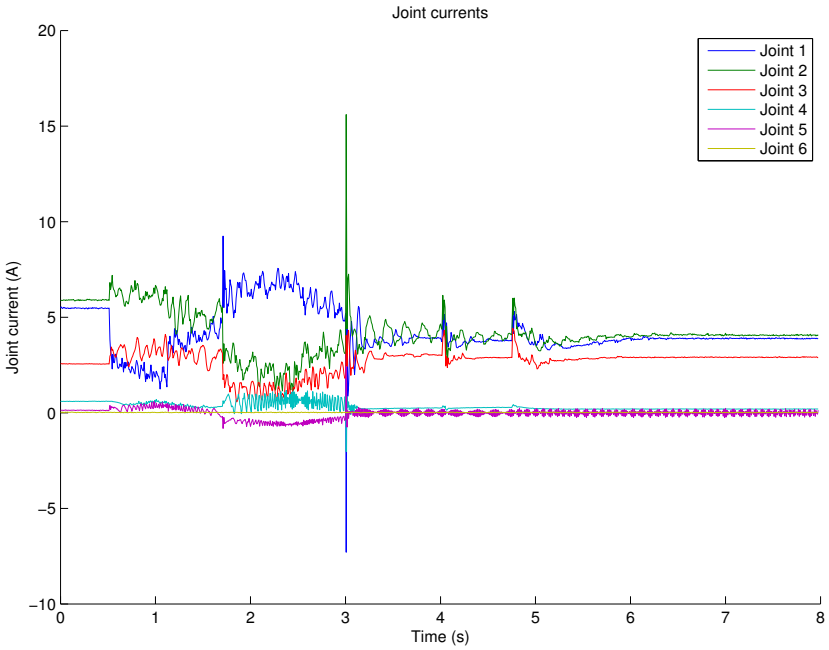

Figure 3.4: Target joint motor currents during a series of overlapping movej commands. One movej command is issued at 0.5 seconds, the next at 1.7s, just before the first finishes, and the third is issued at 3s, way before the second command has finished. Both interruptions lead to acceleration spikes and the second leads to joint torque limit violation.

3.3 The movel command

The code for this test:

```
sleep(0.5)
movel(q1)
sleep(1.2)
movel(q2)
sleep(1.3)
movel(q3)
sleep(5)
```

This command is essentially similar to movej except for including inverse kinematics and, more importantly, inverse dynamics. This creates the ever-bothersome problem of singularities in the dynamics. Apart from that, the same problems exist as when moving with movej.

3.4 The servoj and servoc commands

The code for this test:

```
sleep(0.5)
servoj(q,0,0,0.8)
sleep(1)
```

These command seems to be essentially a less useful version of movej and movel. They do the job of getting from A to B, but the same stopping behaviour as with movec is exhibited, making the use of this command slightly faster and massively more jerky.

3.5 The speedj command

This command seems to be the only command useful for control real-time control of the robot.
The robot accelerates to a set of joint velocities, both velocities and accelerations provided by the user. No combination of interruptions was found that would make the robot attempt accelerations exeeding what was sent in the command.

3.6 The speedl command

For figure 3.5, 3.6 and 3.7, two speedl commands were issued, one at t=0.5s, and then one with a lower speed at t=2s, before the first command finished.

```
sleep(0.5)
speedl(v_1,0.4,1.5)
sleep(1.5)
speedl(v_2,0.4,1.5)
sleep(1.5)
```

It can be seen that the target velocity jumps suddently at t=2s, causing target acceleration to take a proportionally large value. This causes a very large current which triggers joint torque limit violation.
The acceleration limits provided by the command is not enforced. This can be because the trajectory is planned with the limit in mind, but without accounting for current velocity. The reason that the accelerations are heeded in speedj may be that the same assumptions are made and the path is planned accordingly, but since the acceleration limits set by the user is in joint acceleration, it is applied to the joint speeds directly. The same cannot easily be done in the case of speedl.

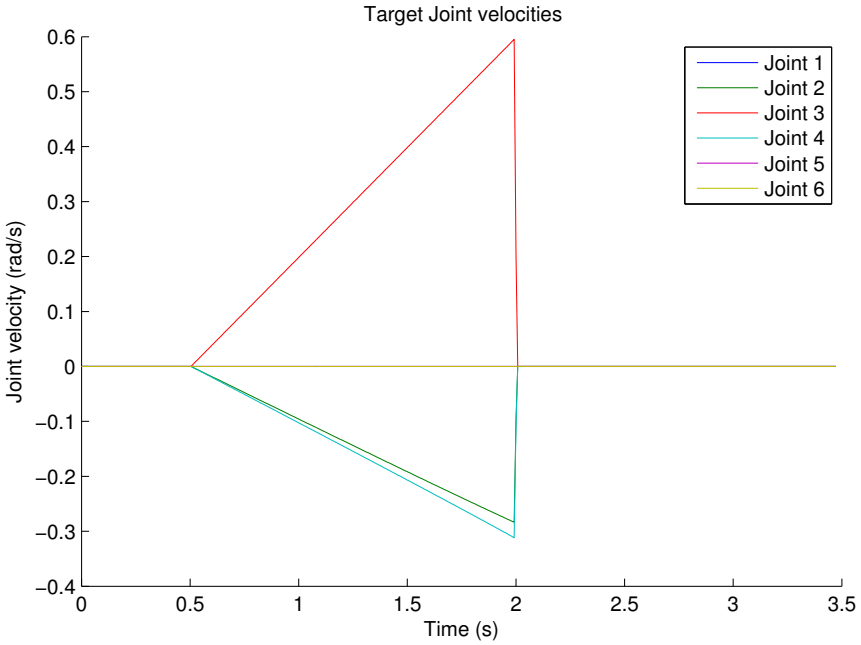

Figure 3.5: Target joint velocities during two overlapping speedl commands

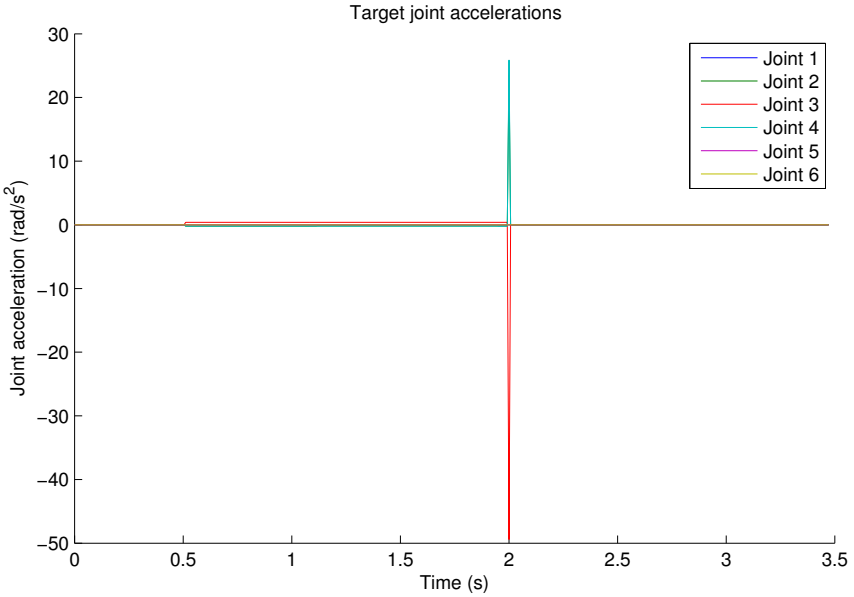

Figure 3.6: Target joint accelerations during two overlapping speedl commands

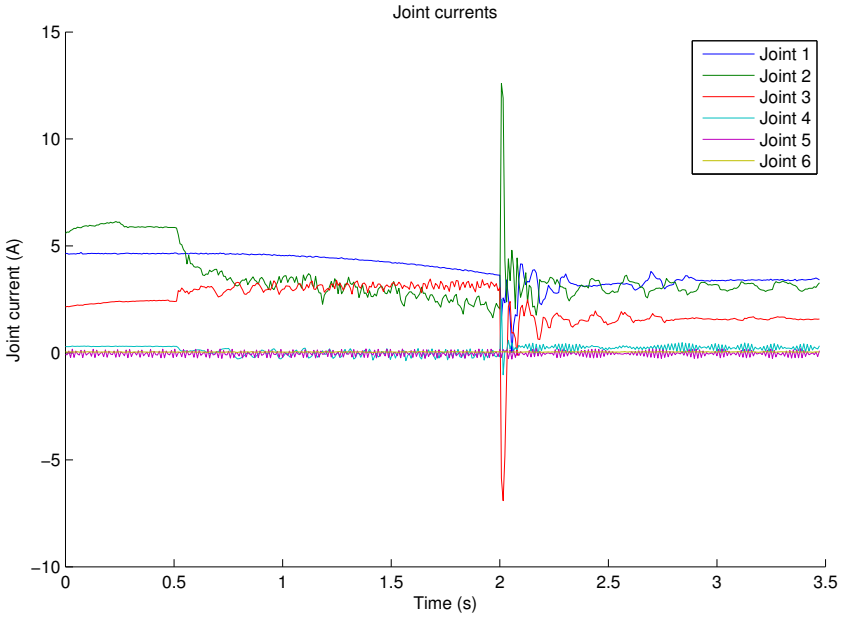

Figure 3.7: Joint currents during to overlapping speedl commands

Issuing many commands in rapid succession with small incremental changes in speed as one would do in a feedback system also induces jerks. Looking at figures 3.8 and 3.9, where a circular speed pattern was sent to the robot, we can see something interesting. The code used was:

```
sleep(0.5)
while t<16:
  pd = [0,0,0,0,0,0]
  pd[0] = pd[0] + 0.1*sin(t)
  pd[2] = pd[2] - 0.1*cos(t)
  speedl(pd,0.1,0.009)
  t = t + 0.008
  sleep(0.008)
end
```

It is generally the case that in order to achieve a specific velocity and rotational velocity in cartesian space, several combinations of joint velocities can be chosen. In this case, it is very likely that each time the inverse dynamics are calculated in order to set the speed in cartesian space, the solution is chosen with some criteria in mind. A sensible criterion would be the lowest sum of joint speeds. Whichever criteria are used do not, it seems, take into account the robot's current dynamic situation. As such, even though the linear

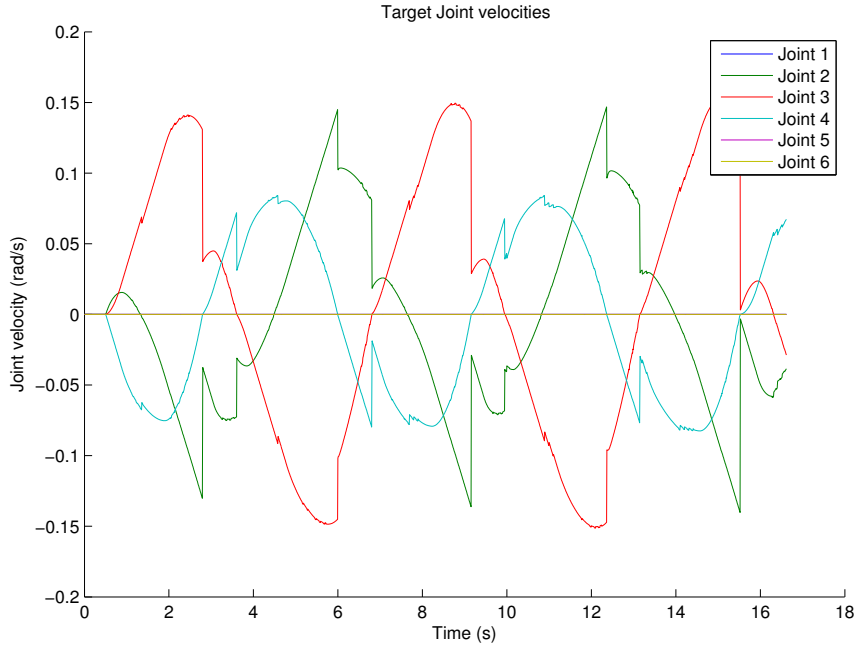

Figure 3.8: Joint velocity targets during an attempted circular motion using speedl

motion would be smooth if the target velocities were achieved, massive (infinite) accelerations would be required to do this. In reality, this is not possible, and the sampling time limits the acceleration so that joint torque limit violation does not happen.

The speedl command is tested again, this time by moving in one direction and changing the velocity every 8 ms by a random amount. This is done to simulate a control input which fluctuates a small amount. The code used was:

```
sleep(0.5)
xd = [0.1,0,0,0,0,0]
for 1 to 375:
  xd = [0.1+randominteger(-1,1)/100.0 ,0 ,0 ,0 ,0 ,0]
  speedl(xd, 0.4, 0.009)
  sleep(0.008)
end
sleep(0.5)
```

The result is shown in Figures 3.10 and 3.11. Here it can be seen that the speedl command handles changes in the target velocity poorly, and it is suspected

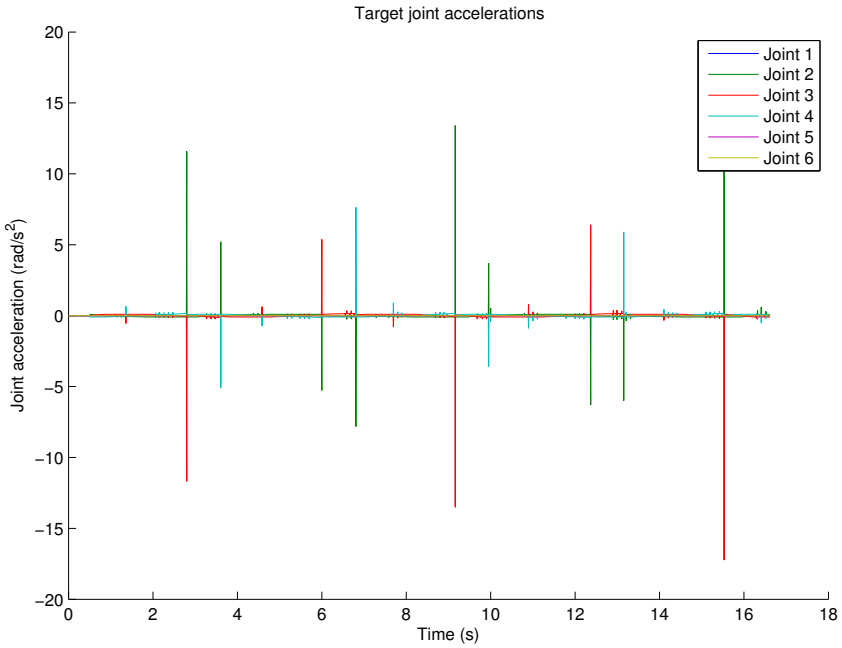

Figure 3.9: Joint acceleration targets during an attempted circular motion using speedl

that the poor behaviour occurs when the acceleration is opposite the current velocity. This behaviour results in the speedl command being unsuitable for real-time control use.

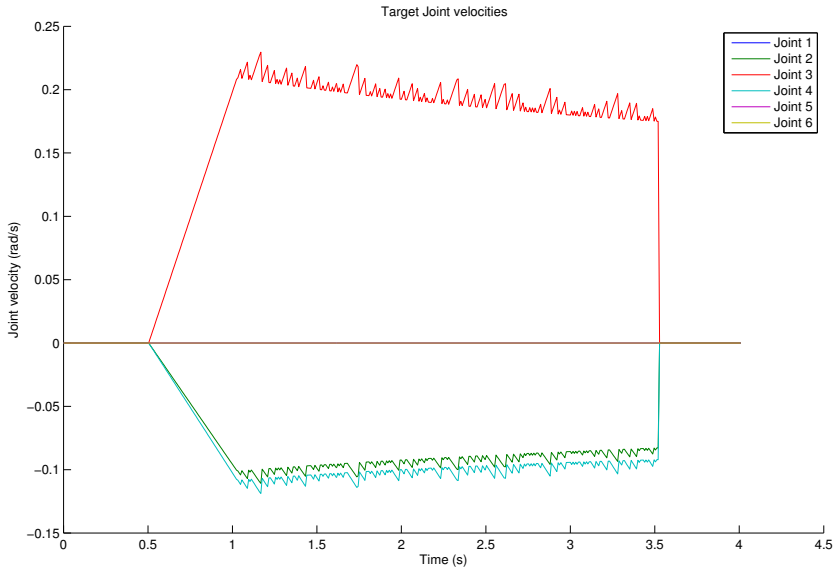

Figure 3.10: Target joint velocities during speedl command where the speed set i changed by a random amount every 8 ms

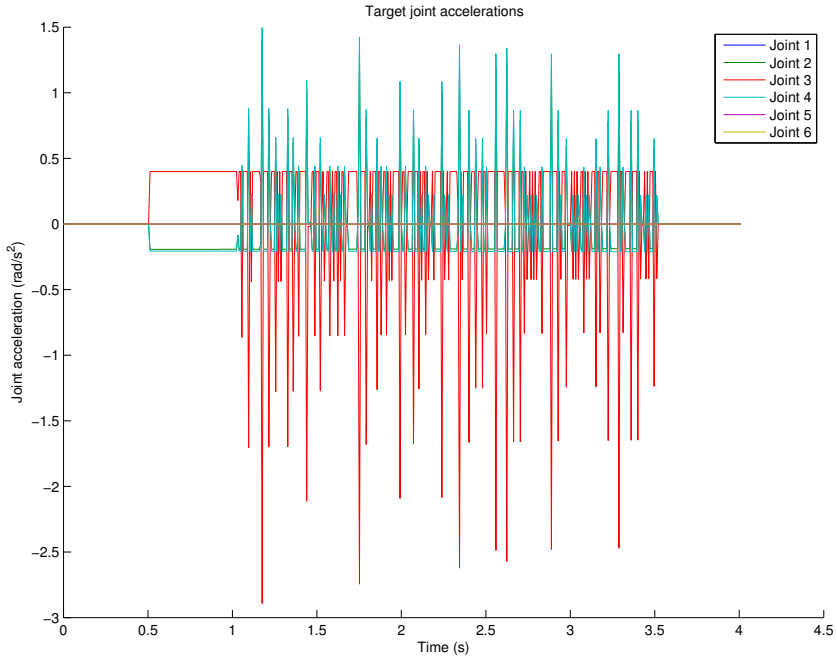

Figure 3.11: Target joint accelerations during speedl command where the speed set i changed by a random amount every 8 ms

3.7 The stopj command

The stopj command decellerates each joint independently to zero with the specified acceleration. The code used was:

```
sleep(0.5)
servoj(q,0,0,0.8)
sleep(0.8)
stopj(1.0)
sleep(0.5)
```

Figures 3.12 and 3.13 illustrate the functionality of the stop commands. It is a good way to stop motions without causing joint torque limit violation.

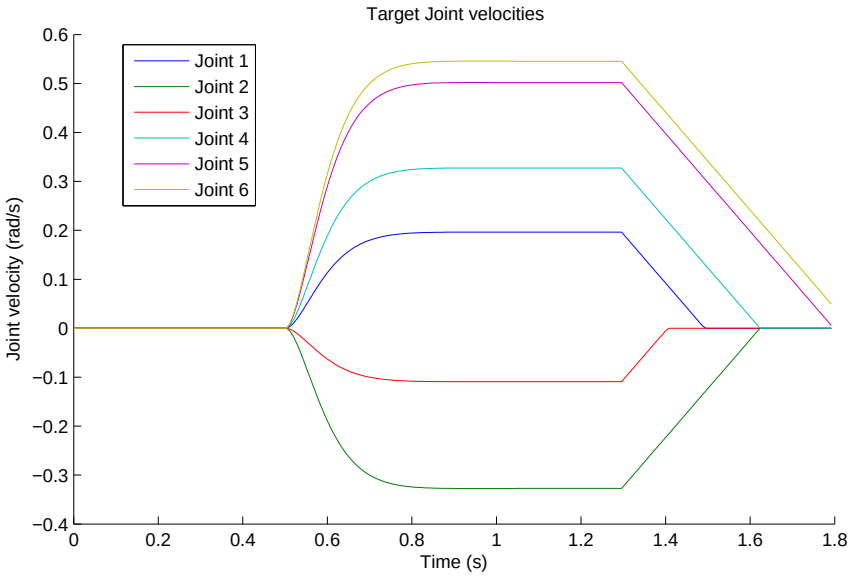

Figure 3.12: Target velocity during execution of servoj command, interrupted by stopj command

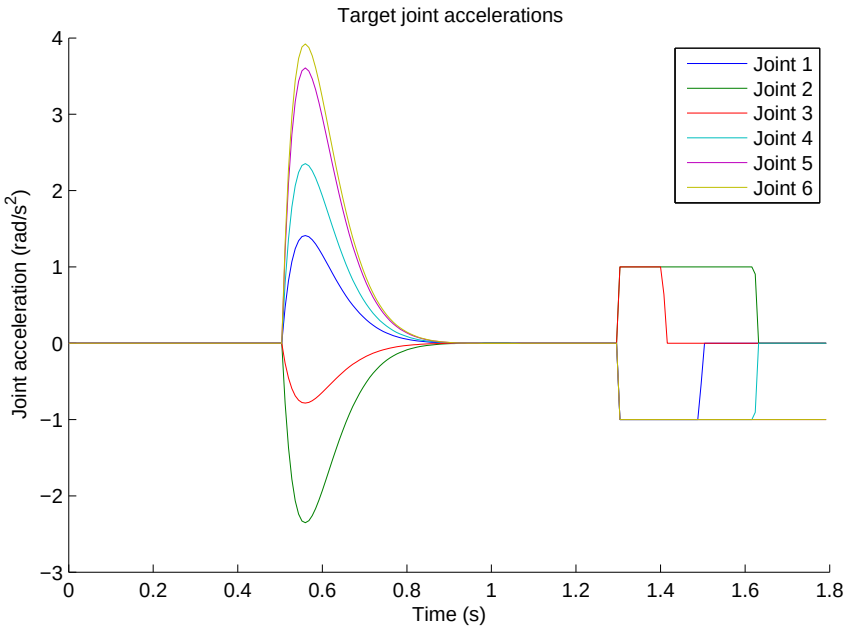

Figure 3.13: Target accelerations during execution of servoj command, interrupted by stopj command

3.8 The stopl command

The stopl command decellerates the TCP to zero with max joint acceleration specified. The code used was:

```
movec(p_via ,p_end ,1 ,0.3)
sleep(2)
stopl(1.0)
sleep(1)
```

Figures 3.14 and 3.15 illustrates the functionality of the stopl command. This commands works as expected, and none of the problems with cartesian space commands seen before is seen when using this command. This points to a problem in the inverse dynamics solution when calculating new trajectories.

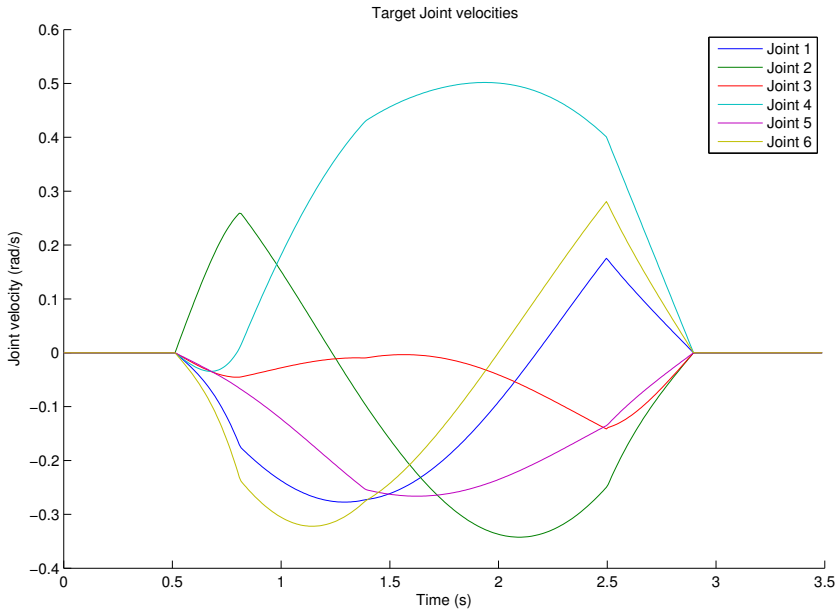

Figure 3.14: Target velocity during execution of movec command, interrupted by stopl command

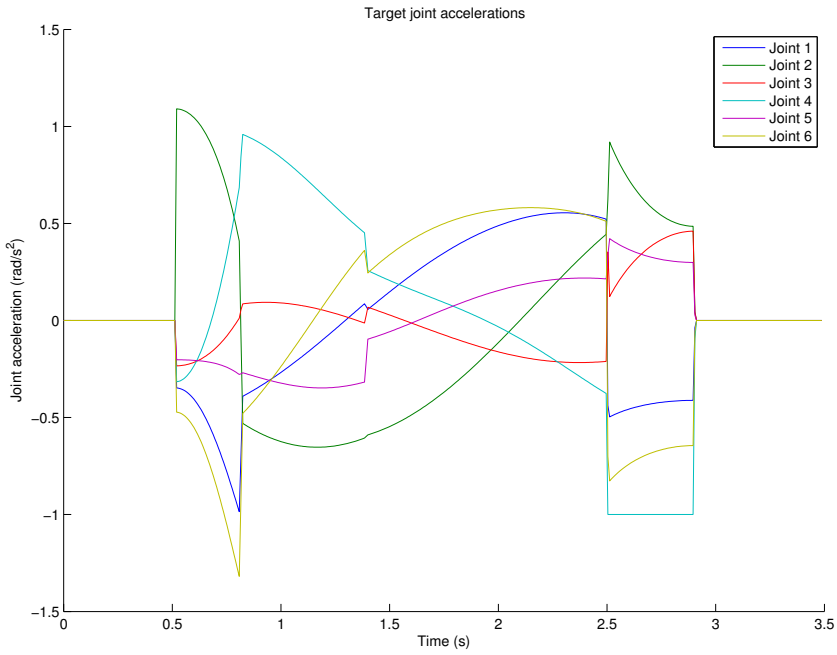

Figure 3.15: Target accelerations during execution of movec command, interrupted by stopl command

4 Conclusion

From tests we have seen that URScript and the URController is designed for open-loop point-to-point control. The only command that can handle interruptions is speedj, all other commands handle interruption poorly. The bandwidth with which one can expect to control the robot is determined by the 24ms round-trip time.